安卓平台的高效云存储系统

许建林　计 13 班　2011011238

1. 应用说明

这个工程是我从 13 年 10 月份以来至今在实验室（网络所崔勇老师的实验室）为一位学长的论文验证所设计实现的原型系统（部分简化版）。主要功能为：加快 client 和 server 之间进行文件传输的速度，通过冗余消除、喷泉码（改进版本，raptor code）等技术来达到这一目的。因为是安卓平台，所以 client 和 server 均为 Java 语言实现。

目前云存储服务风靡全球，这个应用的创新点并不在应用的创意本身，而在于利用了一些较新的技术（冗余消除、raptor code 等）来加快在移动端网络状况较差、设备计算能力一般情况下，文件的传输速度。这两项关键技术将在后面展开介绍。

2. 技术介绍

2.1 冗余消除

Client 上传到 server 上的文件可能部分或者全部都已经存在与 server 端了，找出这部分冗余数据就能减少传输的数据量，从而减少传输时间。

简单的冗余消除包括：

文件粒度：计算整个文件的 hash 值，检查 server 端是否已经存在此文件（百度云的 PC 版本采用的就是此种方法）；

固定大小的文件切块：把文件切分成固定大小的块，计算每一块的 hash 值，判断是否冗余（dropbox 的 PC 版本采用的就是这种方案）。

但是上述方法均有不足之处：如果只在文件前面的部分稍作修改，甚至插入与删除，则切出来的块都会改变，冗余消除的效果就变得很差。

在本系统中采用的是动态切块技术（content defined chunking）[1]。用一个滑动窗口来计算整个文件的内容，当窗口中内容的 hash 值为某一特定值时判定该窗口位置为一个切分点。

---

[1] Muthitacharoen A, Chen B, Mazieres D. A low-bandwidth network file system[C]//ACM SIGOPS Operating Systems Review. ACM, 2001, 35(5): 174-187.

采用这样的切块方案，对文件进行一次修改至多影响到两个块。

在本系统中，我参考原文章中的描述，自己实现了 CDC 切块的代码。

2.1 raptor code[2]

这是一种数据编码方式，简而言之：把原始数据切分为 K 个 SYMBOL，然后把这 K 个 SYMBOL 编码成为更多个 SYMBOL，发送端源源不断的编码并发送这些 SYMBOL，接收端来接收这些 SYMBOL，可以发生丢包，可以发生乱序，当接收端接收到一定数量的 SYMBOL 之后就能以很大的概率解码出原数据。采用这样的编码方式，结合 UDP 进行传输，应用层也不需要实现可靠传输机制（确认与重传机制），就可以完成数据的可靠传输。

在本系统中，我使用了一个开源的 raptor code Java 版本[3]，并对其进行了简单包装。我的代码在工程 Raptor Code 的 raptor.util 包下。

3. 协议设计

整个原型系统采用简单的 client-server 架构，目前的原型系统可以支持多用户同时在线，但是由于投入精力有限，且目标主要集中在如何提高单个文件的传输效率上，所以多用户支持时的系统资源管理（线程、socket 等）很粗糙，自然也就没有考虑 client 身份的认证了。安卓界面也并未进行设计实现，只是用了两个按钮可以进行测试。com.piasy.client 包内的三个类是界面相关，老师可以不用细看。

通信时所有消息都是 json 格式。

3.1 初始化

对于一个用户，首先和 server 的固定端口建立 TCP 链接，该 socket 作为控制通道（ControlChannel），然后在控制通道内进行初始化通信，初始化阶段 client 和 server 的交互过程如图 1 所示：

---

[2] Shokrollahi A. Raptor codes[J]. Information Theory, IEEE Transactions on, 2006, 52(6): 2551-2567.

[3] http://code.google.com/p/raptor-code-rfc/

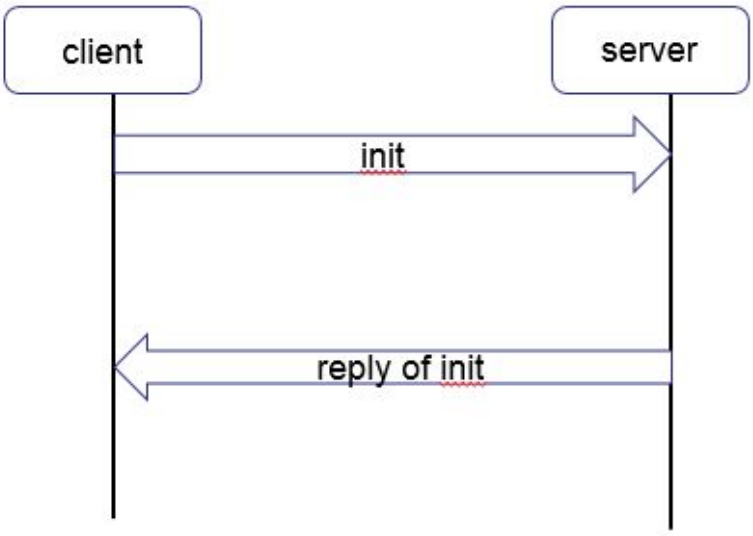


图1：初始化交互

其中 init 包和 reply 包的内容分别如下：

```
{
“type” : “init”,
“name” : name
}
```

```
{
“port” : port
}
```

其中 name 为 client 的用户名，port 为数据通道（DataChannel）的连接端口。后续的文件上传下载请求和响应均在数据通道内进行。

这一交互过程代码分别在 client 和 server 的

void com.piasy.client.controller.Controller.controllerRunnable.new Runnable() {...}.run()

和

void controller.Controller.run()

函数中。

3.2 上传过程

Client 发起上传请求及 server 的响应如图 2 所示：

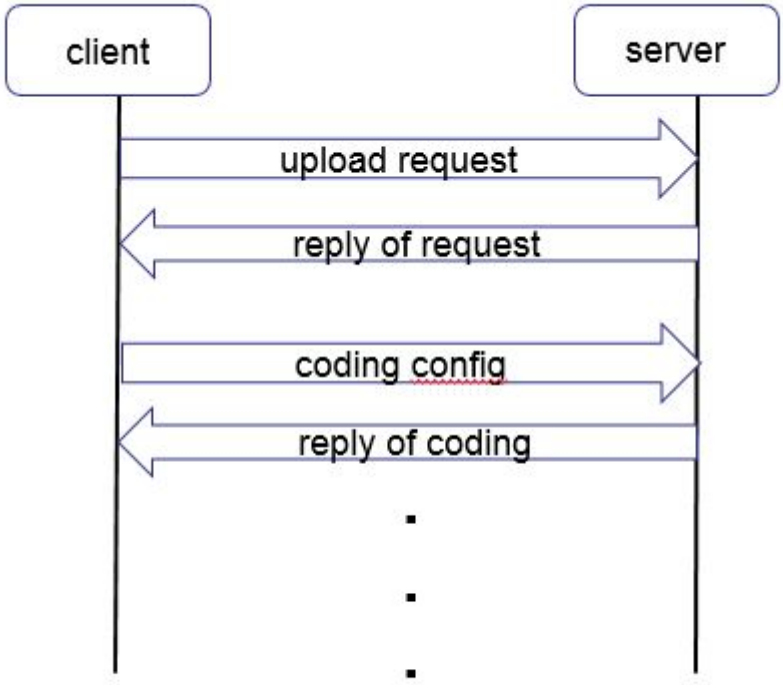


图2：上传过程交互

这些交互都在 DataChannel 中完成，在上传过程中首先采用了 CDC 切块来实现冗余消除，然后对于真正需要传输的数据采用了 raptor code 进行编码传输。

Upload request 和 reply 包格式分别如下：

```
{
“type” : “upload”,
“filename” : filename,
“size” : size,
“blocks” :   [ {
                    “seq” : seq,
                    “size” : size,
                    “hash” : hash
                  }, …
                ]
}
```

```
{
“blocks” : [{
                    “seq” : seq,
                    “action” : action
                }, …
              ]
}
```

其中，filename 为上传的文件名，最外面的 size 为文件大小，blocks 对应的是一个 JSONArray，每个元素为一个 JSONObject，seq 为一个 CDC 切出来的块的序号，size 为其大小，hash 为其 hash 值，String 类型。action 为 String 类型，为”upload”，表示需要上传，或者”success”表示 server 端已经存在不需要上传。

这一次交互的代码在

void com.piasy.client.model.TransferTask.run()

和

void model.DataChannel.TransferThread.upload(JSONObject transferInfo)

函数中。

通过这一次对话，client 即可确定哪些数据需要传输，对于这些数据，采用 raptor code 进行编码，然后使用 UDP 进行传输。因为 raptor code 实际上也是对数据进行了分块，然后单独对每一块数据进行编码，而第一步就已经进行了 CDC 切块，两者切块并不是一致的，所以可能需要把多个 CDC 切的块拼接成一个编码的块，拼接的格式如图 4 所示：

图4：CDC数据块拼接图

number 为这个数据块中的 CDC 数据块数，index 为每一块的编号，size 为其大小，这三个域均为 int 类型，data 即 size 个字节的数据。对于这一块数据进行 raptor code 编码，会有两个编解码的参数需要发给 server，同时 client 还需要使用一个 TCP 连接来进行这一块数据的传输确认。所以这需要进行一次交互，coding config 和 reply 包的内容分别如下：

```
{
“K” : K,
“SYMSIZE” : symsize,
“bsize” : bsize
}
```

```
{
“K” : K,
“SYMSIZE” : symsize,
“bsize” : bsize,
“action” : “start”,
“port” : port,
“ctrport” : ctrport
}
```

这一次的交互在：

void com.piasy.client.model.TransferTask.codingSend(ArrayList<CodingRawData> codingBlocks)

和

void model.DataChannel.TransferThread.upload(JSONObject transferInfo)

函数中。

其中 K 和 symsize 分别为 raptor code 编码时的两个参数，K 值和一个 SYMBOL 的大小，bsize 则为原始数据块的大小（因为原始数据可能小于 K*symsize，编码时会填充一些 0，为了确定填充了多少个 0 以便在解码后还原出原始数据，所以需要把原始数据的大小也发送给

解码方）。port 为这些 SYMBOL 进行 UDP 传输时接收端的端口号，ctrport 则是为了需要在接收端解码成功之后通知发送端，所以需要建立一条 TCP 连接（为了保证确认信息一定到达），ctrport 就是在建立这条连接时 server 监听的端口号。

为了保证 UDP 报文不被分割，所以我们保证每个 UDP 报文为 1KB 左右，而一个 SYMBOL 可能并没有 1KB 大（当然我们用 raptor code 编码时会尽量保证使一个 SYMBOL 为 1KB），所以为了提高 UDP 报文的利用率，我们会把多个 SYMBOL 进行拼接，使其总大小为 1KB 左右，其拼接格式如图 5 所示：

图5：SYMBOL拼接图

因为每个 SYMBOL 的大小 symsize 已经传给了 server，所以这里不需要 size 域。

Server 解码出原始数据需要 K*（1 + overhead）个 SYMBOL，这个数值是 client 与 server 共享的，client 在使用 UDP 发送时，首先会以较快的速度发送这些 SYMBOL，因为 UDP 报文可能丢失，所以剩下的 SYMBOL 也需要继续发送，但以较慢的速度发送。

当 server 接收到了足够多的 SYMBOL 并解码成功之后，就会在新建立的 TCP 连接中发送一个确认包。确认包格式如下：

```
{
“status” : “success”
}
```

这个对话的代码在：

byte[] model.ReceivingThread.decodeOneBlock(DatagramSocket dgserver, JSONObject conf, int blockSeq)

函数中。

收到这条确认消息之后，client 将停止发送 UDP 报文，自此，一个 block 的传输完成，然后继续编码、发送、接收、解码的过程。

同时，对于多核的手机，为了充分利用手机的计算资源，我们采用了多线程机制，多个 block 同时开始编码发送，而且由于对于每一块的编码和传输过程的瓶颈分别是 CPU 和 I/O，实验中也发现编码时间和发送时间基本相同，所以我们又设计了类似于流水线的机制，把编

码和发送分成两个步骤，流水进行。我们定义了一个常量 CODING_THREAD_COUNT，保证每次在编码的线程维持在这个数量，编完码的线程就进入发送阶段开始发送。

3.3 下载过程

下载过程的交互过程和上传过程类似，但是由于 client 端并没有 server 端的大数据集，所以并没有冗余消除的空间，也并未引入多线程机制。交互过程如图 3 所示：

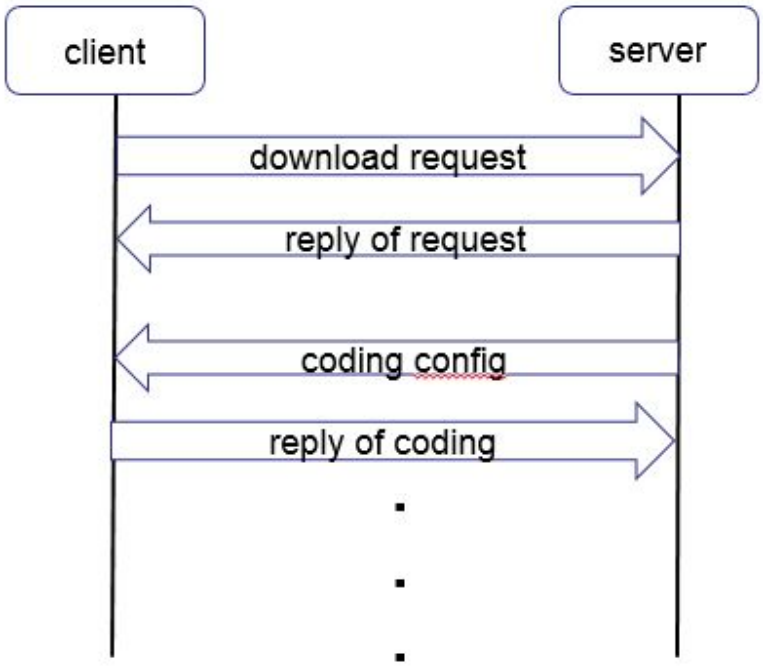


图3：下载过程交互

Download request 和 reply 包格式如下：

```
{
“type” : “download”,
“filename” : filename,
“port” : port
}
```

```
{
“filename” : filename,
“size” : size,
“blocks” : [{
                “seq” : seq,
                “action” : action
            }, ...
          ]
}
```

其中 port 为 client 接收数据的 UDP 连接端口号。

这部分代码在：

void com.piasy.client.model.TransferTask.run()

和

void model.DataChannel.TransferThread.download(JSONObject transferInfo)

函数中。

Coding config 和 reply 包的格式如下：

```
{
“K” : K,
“SYMSIZE” : symsize,
“bsize” : bsize
}
```

```
{
“K” : K,
“SYMSIZE” : symsize,
“bsize” : bsize,
“action” : “start”
}
```

这部分代码在：

boolean model.DataChannel.TransferThread.send(EncodeResult result, int blockSize, int blockSeq)

和

void com.piasy.client.model.TransferTask.download()

函数中。

这里同样也需要对 CDC 数据块进行拼接和对 SYMBOL 进行拼接，与上传过程完全一致。

一个 block 传输完成后，client 也会向 server 发送一个确认包，格式如下：

```
{
“status” : “success”
}
```

这部分代码在：

void com.piasy.client.model.TransferTask.download()

函数中。

4. 实现与测试

4.1 设计模式

client 和 server 端都采用了经典的 MVC 设计模式，而 client 的 Controller 类则采用了单件模式，通过一个静态的方法 getController，在任何代码中都可以获取 Controller 实例，然后通过 Controller 的接口来进行数据处理等操作。

4.2 文件管理

在 server 端，采用文件系统加 mysql 数据库来管理文件（数据），server 端并不会保存完整的文件，保存的最小单位为 CDC 数据块，文件名为其 hash 值。文件信息（metadata）保存在 mysql 数据库中。

Server 端的数据库中有三类表：

（1） blocks 表，结构为：

id	size	hash

每行表示 server 端保存的一个文件块。

（2） files 表，结构为：

id	filename	size	blocknum	owner

每行表示 server 端保存的一个文件。

（3） 每个文件一张表，保存该文件的块信息，表名为<owner>#$%$#<filename>的 hash 值，其结构为：

id	hash	size	blockseq

每行表示该文件的一个块信息。

对数据库的操作我使用了一个类封装了起来，提供需要的操作：查询一个 block 是否存在，添加一个 block，添加一个 file 等。这样使得对 mysql 操作的代码比较集中，方便调试，而且我对一些变量名进行了过滤，避免出现类似 mysql 注入的问题。

4.3 日志记录

为了方便进行调试查错，我设计了一个 Loger 类，专门负责结构化输出日志信息，同时

根据不同的运行模式和日志的等级，有选择的进行记录。

4.4 工程组成

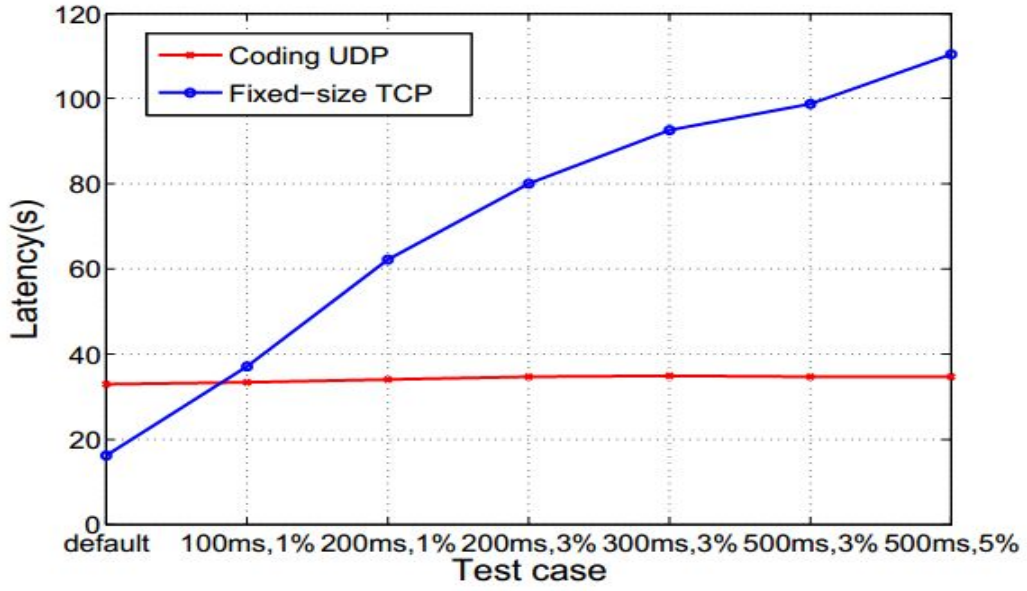
包括简单的注释，自己的代码 3600 余行。

|--Client/src/com/piasy/client/controller	client 的 controller 包
|--Client/src/com/piasy/client/model	client 的 model 包
|--Client/src/com/piasy/client	client 的 viewer 包
|--Server/src/driver	server 的程序入口
|--Server/src/controller	server 的 controller 包
|--Server/src/model	server 的 model 包
|--Server/src/dao	server 的数据库操作封装包
|--Server/src/util	server 的工具包

4.4 性能测试

为了测试这种文件传输策略的效率，我们和 dropbox 的传输策略进行了对比（固定大小的切块，用 TCP 传输，测试代码并不在这个版本中）。

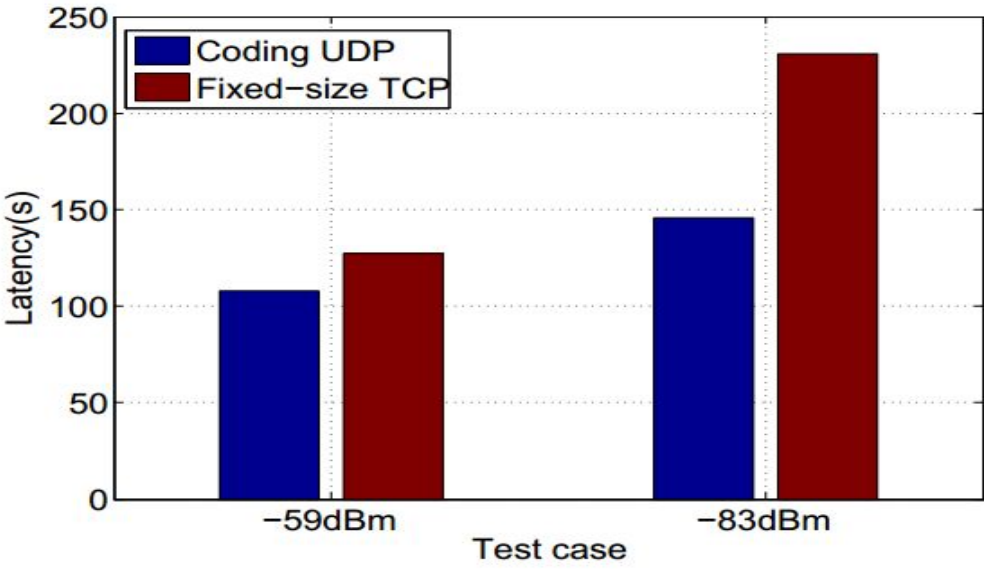
为了体现我们的系统在较差的网络状况下的优势，我们采用了受控的 AP 作为 client 的网络接入点，实验结果如下图所示：



从图中可以看出，我们的系统在网络条件较好时相较于 dropbox 的策略并没有优势，因为网

络传输并没有更快，反而有编码等开销，而当网络条件变差时，我们系统的优势就明显表现出来了，上传时间明显小于 dropbox 策略。

在不同信号强度的 3G 环境下，我们也进行了这样的对比实验，结果如下图所示：



-59dBm 信号强度要优于-83dBm，图中也可以很明显看出，3G 条件下，网络状况较差时，同步时间上我们的策略更加明显。